Introduction

This paper discusses three different approaches, and several variations on them, to solving the problem of scheduling non-identical jobs on a batch processing machine. Batch processing machines, for the purposes of this paper, can process multiple, non-identical jobs simultaneously – but all jobs must be loaded into and unloaded from the machine at once, which introduces a considerable twist on the "simple parallel resources" known from other cases.

The machines in question represents real-life resources like autoclaves or ovens, which can process multiple items at a time, but often cannot be opened at random – in fact, such machines often need to wait for the largest item in the batch to be done before the next batch can be inserted.

1.1 Problem definition

1.2 Organization of this paper

Literature Review

2.1 General MIP and CP

- Who are the "founders" of MIP and CP?
- HP Williams
- Benders

2.2 Batch processing

- Azizoglu
- Dupont
- Sabouni

2.2.1 MIP

- Grossmann

2.2.2 CP

- Baptiste
- Malapert (how does he fit into the picture?)

Modelling the problem

3.1 Bounds on variables known a priori

3.1.1 Lower bound on $L_{\max}$

A lower bound on $L_{\max}$ can be found using the lower bound on the completion date of each bucket q, where a bucket is defined as the set of batches with due date δ_q:

$$L_{\max} \geq C_{\max,q} - \delta_q \quad \forall q \tag{3.1}$$

This works because the buckets up to bucket q are guaranteed to contain all jobs with due dates $d \leq \delta_q$, since the batches within the buckets are ordered by earliest due date (EDD) in the optimal solution. The buckets up to bucket q will likely also contain some later ($d > \delta_q$) jobs in the optimal solution but this does not affect the validity of the lower bound.

Now we need to find $C_{\max,q}$, or at least a lower bound on it, in polynomial time. The simplest approach simply considers the "total elastic area", i.e. the sum of all $s_j p_j$ products:

$$C_{\max,q} \geq \lceil \frac{1}{b} \sum_j s_j p_j \rceil \quad \forall q, \forall \{j \in J | d_j \leq \delta_q\} \tag{3.2}$$

A better lower bound on $C_{\max,q}$ would be given by a preemptive-cumulative schedule. Unfortunately, minimizing $C_{\max}$ for such problems is equivalent to solving a standard bin-packing problem, which requires exponential time.

3.1.2 Upper bound on $L_{\max}$

An upper bound on $L_{\max}$ can be found by using a dispatch rule to find a feasible, if not optimal, schedule. A good approach could be the "best-fit" heuristic proposed in the original paper. This has not been implemented yet.

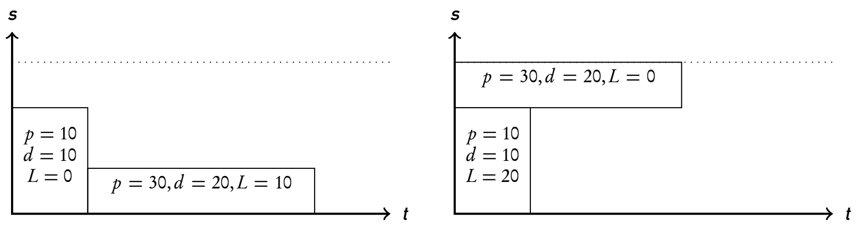

Figure 3.1: Overzealous batch elimination can increase $L_{\max}$

3.1.3 Bounding the number of batches n_k

Initially, the number of batches needed is assumed to be equal to the number of jobs: $n_k = n_j$. Reducing n_k by pre-computing the maximum number of batches needed shrinks the x_{jk} matrix, and prunes potential search branches in branch-and-bound decomposition approaches.

Unfortunately, we cannot make a general statement that optimal solutions never have more batches than other feasible solutions – a simple counterexample is shown in figure 3.1.[1]

Starting out with a one-job-per-batch schedule sorted by EDD, we can explore all feasible batch configurations recursively. To generate any other feasible schedule (including the optimal solution), jobs j are rescheduled ("moved back") from their original batch k_{origin} into a prior batch k_{earlier}. This eliminates k_{origin} and requires, of course, that k_{earlier} has sufficient capacity. If the job's processing time p_j exceeds that of k_{earlier}, then the lateness of batches between k_{earlier} and k_{origin} will increase; the merit of such a move cannot be judged in prior, so it is an *unsafe* batch elimination. *Safe* eliminations, on the other hand, will never worsen $L_{\max}$, and only they can be considered when bounding n_k a priori.

Algorithm 1 outlines a recursive method to find an upper bound on n_k, recognizing safe batch eliminations only.

This algorithm evidently requires exponential time. A relaxed variant is a possible option: if only a single unsafe move *into* a batch is possible, no safe eliminations into that batch are considered at all and we skip to the next batch. This would greatly speed up the recursion but also significantly weaken the usefulness of the resulting upper bound.

In a brach-and-bound decomposition approach in which batches are modelled in-

[1]To be more precise, we cannot state that at least one optimal solution is in the subset of feasible solutions that uses the fewest number of batches – a dominance situation that could be exploited, were it true.

Algorithm 1 Recursive algorithm to find an upper bound on n_k

```
if no open jobs left then                ▷ if this is a "leaf node" in the recursion
    update UB($n_k$)
end if
find the combination of unsafe later jobs that fills up the capacity most, leaving us with capacity $b_r$
find all combinations $x$ of safe later jobs that fit into $b_r$
repeat
    $x$ = next safe job combination
    ignoreJobs ← $x$                ▷ make the moves, let the next recursion level deal with the rest of the jobs
    spawn and run child node with $J \setminus$ ignoreJobs
    ignoreJobs ← ignoreJobs $\setminus x$
until all combinations have been explored
return
```

dividually, an upper bound on n_k could be used to limit the depth of the search tree, or, in combination with a method to determine a lower bound on the remaining jobs' n_k at every node, to actively prune the search tree during the search. The latter method, however, would also run in exponential time as it, again, would require knapsack-type reasoning unless we use a much less powerful relaxation.

3.2 MIP model

Malapert's original MIP approach is given in Model 3.2.1.

The constraint 3.8 is the EDD constraint.

3.2.1 Grouping empty batches

The given formulation lacks a rule that ensures that no empty batch is followed by a non-empty batch. Empty batches have no processing time and a due date only bounded by $d_{\max}$, so they can be sequenced between non-empty batches without negatively affecting $L_{\max}$. Since, however, desirable schedules have no empty batches scattered throughout, we can easily reduce the search space by disallowing such arrangements. The idea is illustrated in Figure 3.2.

A mathematical formulation is

$$\sum_{j \in J} x_{j,k-1} = 0 \rightarrow \sum_{j \in J} x_{jk} = 0 \quad \forall k \in K. \tag{3.11}$$

To implement this, we can write constraints in terms of an additional set of binary variables, e_k, indicating whether a batch k is empty or not:

$$\sum_{k\in K} x_{jk} = 1 \qquad \forall j \in J \tag{3.3}$$
$$\sum_{j\in J} s_j x_{jk} \leq b \qquad \forall k \in K \tag{3.4}$$
$$p_j x_{jk} \leq P_k \qquad \forall j \in J, \forall k \in K \tag{3.5}$$
$$C_{k-1} + P_k = C_k \qquad \forall k \in K \tag{3.6}$$
$$(d_{max} - d_j)(1 - x_{jk}) + d_j \geq D_k \qquad \forall j \in J, \forall k \in K \tag{3.7}$$
$$D_{k-1} \leq D_k \qquad \forall k \in K \tag{3.8}$$
$$C_k - D_k \leq L_{\max} \qquad \forall k \in K \tag{3.9}$$
$$C_k \geq 0, P_k \geq 0 \text{ and } D_k \geq 0 \qquad \forall k \in K \tag{3.10}$$

Model 3.2.1: Malapert's original MIP model

$$e_k + \sum_{j\in J} x_{jk} \geq 1 \qquad \forall k \in K, \tag{3.12}$$
$$n_j(e_k - 1) + \sum_{j\in J} x_{jk} \leq 0 \qquad \forall k \in K. \tag{3.13}$$

Constraints (3.12) enforce $e_k = 1$ when the batch k is empty. Constraints (3.13) enforce $e_k = 0$ otherwise, since the sum term will never exceed n_j. The rule 3.11 can

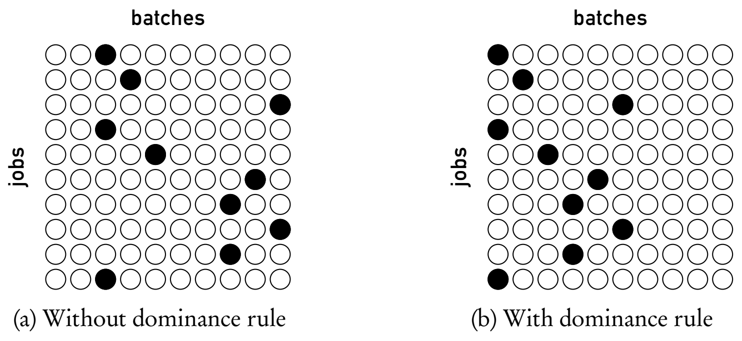

Figure 3.2: Dominance rule to eliminate empty batches followed by non-empty batches

now be expressed as $e_{k-1} = 1 \rightarrow e_k = 1$, and implemented as follows:

$$e_k - e_{k-1} \geq 0 \quad \forall k \in K. \tag{3.14}$$

We can also prune any attempts to leave the first batch empty by adding a constraint $e_0 = 0$.

This, while it works just fine, actually takes longer than without it.

3.2.2 No postponing of jobs to later batches

Since the jobs are already sorted by non-decreasing due dates, it makes sense to explicitly instruct the solver never to attempt to push jobs into batches with a greater index than their own: even if every job had its own batch, it would be unreasonable to ever postpone a job to a later batch.

$$x_{jk} = 0 \quad \forall \{j \in J, k \in K | j > k\} \tag{3.15}$$

3.3 CP model

The mixed integer model can be turned into a constraint programming model with just a small number of modifications.

3.3.1 Bin-packing and cumulative constraints

This makes sure the jobs are distributed into the batches such that no batch exceeds the capacity b:

$$\texttt{pack}(J, K, b) \tag{3.16}$$

The cumulative constraint functions similarly, but instead of packing discrete bins, it enforces non-overlapping constraint on the temporal (`IntervalVariable`) J variables.

$$\texttt{cumul}(J, b) \tag{3.17}$$

3.3.2 Temporal constraints

These constraints are implemented using `IntervalVar` objects, which offer properties such as `lengthOf` or `endOf`.

$$P_k \geq \max_{j:B_j=k} p_j \qquad \forall k \in K \tag{3.18}$$

$$D_k \leq \min_{j:B_j=k} d_j \qquad \forall k \in K \tag{3.19}$$

$$C_k + P_{k+1} = C_{k+1} \qquad \forall k \in K \tag{3.20}$$

$$L_{\max} \geq \max_k (C_k - D_k) \tag{3.21}$$

The first constraint ensures that each batch is as long as its longest job. The second constraint ensures that the earliest-due job sets the due date. The third constraint defines batch completion dates, and the fourth constraint defines $L_{\max}$.

The first two temporal constraints exploit the temporal features of `IntervalVars`.

3.3.3 Constraint on number of batches with length $P_k > p$

Since batches take on the processing time of their longest job, there is at least one batch with $P = \max_j p_j$:

$$\texttt{globalCardinality}(|P_k = \max_j p_j| = 1) \tag{3.22}$$

We can proceed to fill batches with jobs, ordered by non-increasing processing time, based on algorithm 2.

At the end of this algorithm, we can state:

$$\texttt{globalCardinality}(|P_{k-1,\min} > P_k \geq P_{k,\min}| \geq 1) \quad \forall k \in \{k_0, \dots, k_{LB(n_k)}\} \tag{3.23}$$

The algorithm sorts jobs by non-increasing p, and then fills batches job by job. If a job fits into a previous batch, it is assigned there. If a job fits into multiple previous batches, it is assigned to the batch with the smallest remaining capacity. This is sometimes called *best-fit dereasing* rule, and works as follows: let $J^\star$ be the set of jobs sorted by p, then at least one batch will be as long as the longest job $j^\star_1$. If the next n jobs fit into this batch, then there is at least one batch not shorter than $j^\star_{n+1}$, and similarly for subsequent batches. Unfortunately, optimal solution may perform better than the packing heuristic in terms of “vertical” (s_j) bin packing, and may thus require fewer batches. We therefore need to find a lower bound $LB(n_k)$ on the number of batches, and we can only guarantee the first $LB(n_k)$ of the above constraints to hold in the optimal solution. Finding a true lower bound is a two-dimensional bin packing problem, which runs in exponential time … so we have to

Algorithm 2 Generating lower bounds on batch lengths

```
$J^\star \leftarrow J$                                            ▷ initialize all jobs as unassigned jobs
$n_k \leftarrow 1$; $S_k \leftarrow \{0\}$; $P_{k,\min} \leftarrow \{0\}$        ▷ Create one empty batch of size and length zero
sort $J^\star$ by processing time, non-increasing
repeat
    $j \leftarrow J^\star$.pop()                                   ▷ select job for assignment, longest job first
    loop through all $n_k$ existing batches $k$, first batch first
        $k_p \leftarrow \varnothing$                                ▷ no feasible batch
        $c_{\min} = b$                                 ▷ currently known minimum remaining capacity
        if $s_j < b - S_k$ and $b - S_k < c_{\min}$ then
            $k_p \leftarrow k_p$; $c_{\min} \leftarrow b - S_k$
        end if
    end loop
    if $|k_p| = 1$ then
        $S_{k_p} \leftarrow S_{k_p} + s_j$                          ▷ assign job $j$ to batch $k_p$
    else
        if $n_k < LB(n_k)$ then
            $n_k \leftarrow n_k + 1$                               ▷ open new batch
            $S_{n_k} \leftarrow s_j$; $P_{n_k,\min} \leftarrow p_j$          ▷ assign $s_j$ and $p_j$ to the new batch
        else
            leave the loop now and end.
        end if
    end if
until $J^\star$ is empty
```

come up with an even lower bound – right now I can only think of j_0, the number of jobs ordered by decreasing s_j that can never fit into a batch together.

Furthermore, if all jobs have different processing times, all batches will have different processing times as well: `alldifferent`(P_k). If m out of n_j jobs have different processing times, we can still enforce `k_alldifferent`(P_k, m). Propagation rules for this constraint were given in [?]. I can't find anything on this w.r.t. CP Optimizer, so I may have to implement this myself … time permitting.

3.3.4 Temporal constraints on a job's start date

Given any partial assignment of jobs and an open job j, we can reason that

- if the first batch with a due date later than the job is k, then the job cannot be part of a batch after k – this would result in a non-EDD sequence of batches.
- if the first batches up to $k-1$ offer not enough capacity for j due to the given partial assignment, then the job cannot be part of a batch before k.

Since batches are *not* dynamically created like in Malapert's solution but fixed from the start, any partial assignment that fails due to these constraints cannot be part of an optimal solution.

This constraint is redundant with both the $(C_{k+1} \geq C_k)$ and `packing` constraints, but may help accelerate the propagation in some cases.

3.3.5 Grouping empty batches

Just like in the MIP model, we can force empty batches to the back and thus establish dominance of certain solutions. The implementation is much easier than in the MIP model:

$$\texttt{IfThen}(C_{k+2} > C_k, C_{k+1} > C_k) \quad \forall k \in \{k_1, \ldots, k_{n_k-2}\} \tag{3.24}$$

3.3.6 No postponing of jobs to later batches

Just like in the MIP model, jobs should never go into a batch with an index greater than their own:

$$x_{jk} = 0 \quad \forall j, k : j > k \tag{3.25}$$

This is implemented as $\texttt{assignments}_j \leq k$.

This constraint is analogous to the concept of finding an upper bound on n_k – essentially, it would be very helpful to find an upper bound on the latest batch for *every* job.

3.4 Decomposition approach

Instead of solving the entire problem using one model, we can solve the problem step by step, using the best techiques available for each subproblem. This approach is inspired by a method called *Benders' decomposition*.

3.4.1 Branch-and-bound by batch in chronological order

A basic version of this approach uses branch-and-bound to transverse the search tree. At each node on level ℓ, a single MIP and/or CP model is run to assign jobs to batch $k = \ell$. The remaining jobs are passed to the children nodes, which assign jobs to the next batch, and so on – until a solution, and thus a new upper bound on $L_{\max}$, is found. Several constraints are used to prune parts of the search tree that are known

$$\begin{tikzcd}
 & & & \text{MIP} \arrow[dll] \arrow[d] \arrow[drr] & & & & k=1 \\
 & \text{MIP} \arrow[dl] \arrow[dr] & & \text{MIP} & & \text{MIP} \arrow[dl] \arrow[dr] & & k=2 \\
\text{MIP} & & \text{MIP} \arrow[d] & & \text{MIP} \arrow[d] & & \text{MIP} & k=3 \\
 & & \text{Solution 1} & & \text{Solution 2} & & &
\end{tikzcd}$$

Figure 3.3: Batch-by-batch decomposition using MIP only

to offer only solutions worse than this upper bound. Figure 3.3 shows an example in which a MIP model is used to assign jobs to the batch at every node. Algorithm

Algorithm 3 MIP node class code overview

```
update *currentAssignments*                          ▷ this keeps track of where we are in the tree
**if** no jobs given to this node **then**     ▷ if this is a leaf node, i.e. all jobs are assigned to batches
    calculate $L_{\max,\text{current}}$ based on *currentAssignments*
    **if** $L_{\max,\text{current}} < L_{\max,\text{incumbent}}$ **then**
        $L_{\max,\text{incumbent}} \leftarrow L_{\max,\text{current}}$
        *bestAssignments* ← *currentAssignments*
    **end if**
    return to parent node
**end if**
set up MIP model                                                        ▷ as described below
**repeat**
    $x_j \leftarrow$ model.solve($x_j$)                        ▷ let model assign jobs to the batch
    spawn and run child node with all $\{j|x_j = 0\}$          ▷ pass unassigned jobs to children
    add constraint to keep this solution from recurring    ▷ this happens once the child node returns
**until** model has no more solutions
return to parent node
```

3 outlines what happens at each node: the model finds the best jobs to assign to the batch according to some rule, lets the children handle the remaining jobs, and tries the next best solution once the first child has explored its subtree and backtracked.

Using MIP and cumulative packing after the batch

The first version of the branch-and-bound batch-by-batch decomposition uses a MIP model at each node to assign jobs to the respective batch. The remaining jobs are packed such as to minimize their $L_{\max}$, with a relaxation of the batching requirement, i.e., as if on a cumulative resource.

$$
\begin{array}{lllr}
\text{Min.} & L_{\text{max,cumul}} & & (3.26)\\
\text{s. t.} & \sum_j s_j x_j \le b & \forall j \in J & (3.27)\\
& P_k \ge p_j x_j & \forall j \in J & (3.28)\\
& \sum_j x_j \ge 1 & \forall \{j \in J | d_j = \min(d_j)\} & (3.29)\\
& P_k + \frac{1}{b}\sum_i s_i p_i \le d_j + L_{\text{max,incmb}} - 1 - v_k & \forall j \in J, \forall \{i \in J | d_i \le d_j\} & (3.30)\\
& \sum_t u_{jt} = 1 & \forall j \in J & (3.31)\\
& \sum_j \sum_{t' \in T_{jt}} u_{jt'} \le b & \forall t \in \mathscr{H} & (3.32)\\
& (v_k + t + p_j) u_{jt} \le d_j + L_{\text{max,incmb}} - 1 & \forall j \in J, \forall t \in \mathscr{H} & (3.33)\\
& L_{\text{max,cumul}} \ge (v_k + t + p_j) u_{jt} - d_j & \forall j \in J, \forall t \in \mathscr{H} & (3.34)\\
& u_{j,t=1} = x_j & \forall j \in J & (3.35)\\
& u_{it} \le (1 - x_j) & \forall i, j \in J, \forall t \in \mathscr{H} \setminus \{2\} & (3.36)\\
& b - \sum_j s_j x_j \le (b w_j + 1) s_j & \forall j \in J & (3.37)\\
& P_k + 2 w_j n_t \ge p_j + n_t x_j & \forall j \in J & (3.38)\\
& P_k - 2(1 - w_j) n_t \le p_j + n_t x_j - 1 & \forall j \in J & (3.39)
\end{array}
$$

Model 3.4.1: MIP model in batch-by-batch branch-and-bound

Model 3.4.1 implements a time-indexed cumulative constraint on the non-batched jobs. Constraints (3.27) through (3.29) ensure that the batch stays below capacity, define the duration of the batch P_k and force at least one of the earliest-due jobs into the batch.

Constraints (3.30) express the interval relaxation used to ensure that no jobs exceed their latest allowable finish date given any batch assignment. Even jobs that are assigned to the batch have to fulfill this requirement.

x_j	is 1 iff job j is assigned to the batch
u_{jt}	is 1 iff job j starts in time slot t
T_{jt}	is the set of all time slots occupied by job j if it ended at time t, that is $T_{jt} = \{t - p_j + 1, \ldots, t\}$
v_k	is the start time of the batch at the given node in the search tree
$L_{\max,\text{incmb}}$	is the incumbent (known best) value of and thus an upper bound on $L_{\max}$
$\mathscr{H}$	is the set of all indexed time points $\{1, \ldots, n_t\}$
w_j	is 1 iff job j is either longer than the batch ($p_j > P_k$) or already part of the batch ($x_j = 1$)

Table 3.1: Notation used in the decomposition model

Constraints (3.31) and (3.32) implement the cumulative nature of the post-batch assignments by ensuring that each job starts only once, and no jobs overlap on a given resource at any time (for the purposes of this model, the batch machine is considered divisible into b unary resources).

Constraints (3.33) again limit the possible end dates of a job, but unlike (3.30), they use the time assignments on the cumulative resource to determine end dates. Constraints (3.34) define the value of $L_{\max,\text{cumul}}$, the maximum lateness of any job in the non-batched set.

Constraints (3.35) force batched jobs to start at $t = 0$, while (3.36) force *all* jobs to start either at $t = 0$ or after the last batched job ends.

Constraints (3.37) enforce a dominance rule: jobs must be assigned to the batch such that the remaining capacity, $b_r = b - \sum_j s_j x_j$, is less than the size s_j of the *smallest* job from the set of non-batched jobs that are *not longer* than the current batch. That is, if there exists an non-batched job j with $p_j \leq P_k$ and $s_j \leq b_r$, then the current assignment of jobs is infeasible in the model. The reasoning goes as follows: given any feasible schedule, assume there is a batch k_a with b_r remaining capacity and a later batch k_b containing a job j such that $s_j \leq b_r$ and $p_j \leq P_{k_a}$. Then job j can always be moved to batch k_a without negatively affecting the quality of the solution: if the schedule was optimal, then moving j will not affect $L_{\max}$ at all (otherwise, it was no optimal schedule); if the schedule was not optimal, then $L_{\max}$ will stay constant (unless j was the longest job in k_b and $L_{\max}$ occured in or in a batch after k_b, in which case $L_{\max}$ will be improved).

This rule is implemented by means of a binary variable w_j, which, as defined by constraints (3.37) and (3.38), is 1 iff $p_j > P_k \vee x_j = 1$. These are the cases in which a job j is *not* to be considered in (3.36), and so w_j is used to scale s_j to a value insignificantly large in the eyes of the constraint's less-than relation.

After a solution is found, a child node in the search tree has run the subtree and returned, a constraint of the form

$$\sum_j x_j + \sum_i (1 - x_i) \leq n_j - 1 \quad \forall\{j \in J | x_j = 1\}, \forall\{i \in J | x_i = 0\} \tag{3.40}$$

is added to the model before the solver is called again, to exclude the last solution from the set of feasible solutions.

Using CP and cumulative packing after the batch

This approach is equivalent, but we now use CP to select the batch assignments, again based on a minimized $L_{\max}$ among the non-batched jobs.

$$\begin{aligned}
\text{Min.} \quad & L_{\max} && && (3.41)\\
\text{s.t.} \quad & \texttt{IfThen}(x_j = 0, \texttt{startOf}(j) \geq P_k) && \forall j \in J && (3.42)\\
& \texttt{IfThen}(\texttt{startOf}(j) \geq 2, x_j = 0) && \forall j \in J && (3.43)\\
& \texttt{IfThen}(x_j = 1, \texttt{startOf}(j) = 0) && \forall j \in J && (3.44)\\
& \texttt{IfThen}(\texttt{startOf}(j) = 1, x_j = 1) && \forall j \in J && (3.45)\\
& P_k \geq x_j p_j && \forall j \in J && (3.46)\\
& \texttt{endOf}(j) = d_j + L_{\max,\text{incmb}} - 1 && \forall j \in J && (3.47)\\
& L_{\max} \geq \texttt{endOf}(j) - d_j && \forall j \in J && (3.48)\\
& \texttt{IfThen}(p_j \leq P_k \wedge x_j = 0, b - \sum_{j \in J} s_j x_j \leq s_j) && \forall j \in J && (3.49)\\
& P_k + \frac{1}{b} \sum_i s_i p_i \leq d_j + L_{\max,\text{incmb}} - 1 - v_k && \forall j \in J, \forall\{i \in J | d_i \leq d_j\} && (3.50)\\
& \sum_j x_j \geq 1 && \forall\{j \in J | d_j = \min(d_j)\} && (3.51)\\
& \texttt{cumul}(J, b) && && (3.52)
\end{aligned}$$

Model 3.4.2: CP model in batch-by-batch branch-and-bound

3.4.2 Potential improvements

Improve the initial $L_{\text{max,incumbent}}$ A better initial upper bound on L_{max} can help prune some branches of the search tree from the outset. There are several dispatch rules (or maybe other heuristics?) that could be explored to do this better.

Improve $L_{\text{max,incumbent}}$ during search It may be useful to use a heuristic like above to "complete the schedule" once a promising partial schedule has been generated. I have yet to identify situations where this is always helpful.

3.5 Empirical results

Malapert uses benchmark data by Daste. For design purposes, I created my own set of randomized job lists, with $s_j, p_j \in [1,20]$ and $d_j \in [1,10n]$ where n is the number of jobs. Here is the Python code used:

```
import random

random.seed(None)

for i in range(11, 20) + range(10,100,10):
    for j in range(1, 10):
        with open( ”data_” + str(i) + ”_” + str(j), ”wb”)
                    as csvfile:
            csvwriter = csv.writer(csvfile, delimiter=”,”)
            csvwriter.writerow([’s_j’, ’p_j’, ’d_j’])
            for job in range(i):
                csvwriter.writerow([random.randrange(1,20),
                                    random.randrange(1,20),
                                    random.randrange(1,5*i)])
        csvfile.close()
```

CSV files can be read in with `IloCsvReader` and `IloCsvLine`.

3.5.1 Comparison of CPU time used by various models

Figure 3.4 shows the CPU time used by the models. Ten sample job sets are used per number of jobs, represented as $[\{10.0,\ldots,10.9\},\{11.0,\ldots,11.9\},\ldots]$ on the horizontal axis. The models were run on an i7 Q740 CPU, which has 2×4 cores. Cplex supports multithreading and as such, elapsed wall clock time is significantly lower in most cases.

The CP branch-and-bound model times out on most instances and is not shown here.

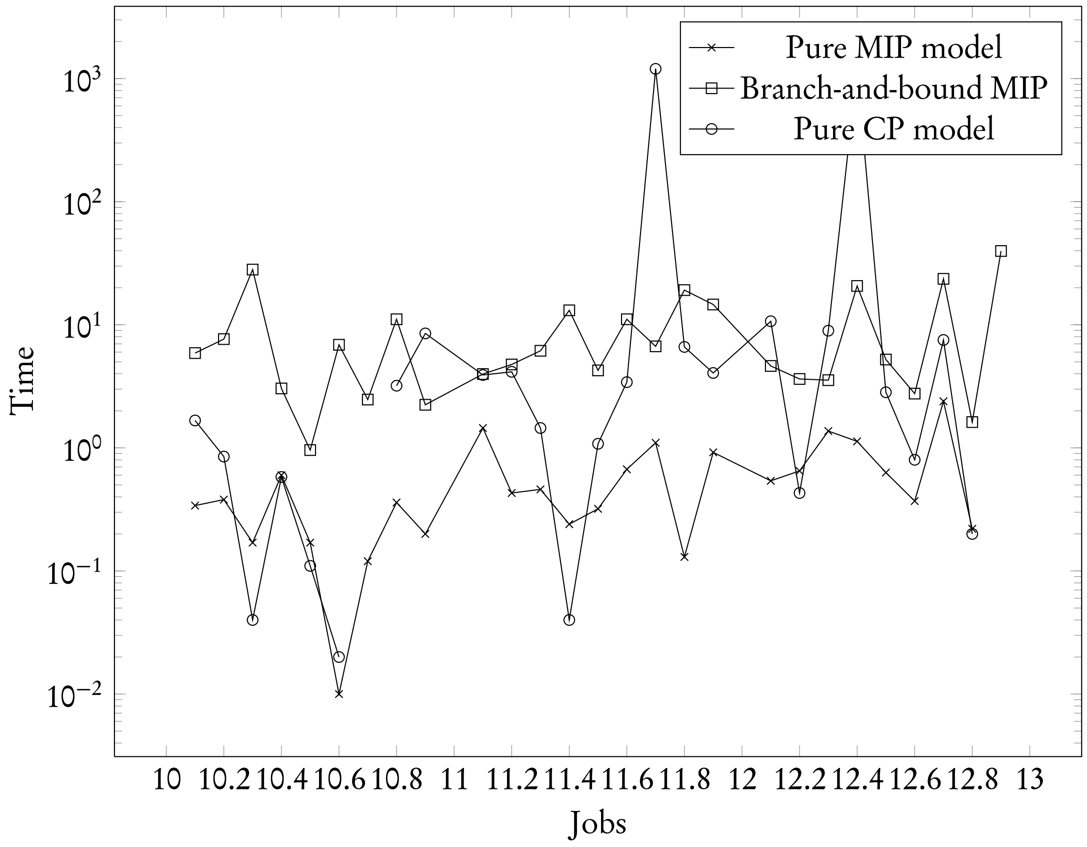

Figure 3.4: Comparison of CPU time used by different models.